Brno University of Technology
Faculty of Information Technology

Microprocessors and Embedded Systems

# ESP8266: LED Control (IoT, WiFi Access Point for Cell Phones)

December 30, 2018

Martin Smutný (xsmutn13)

# Contents

# 1 Introduction

The purpose of this project is to design and implement (using Arduino IDE [3]) an Embedded System, that enables a client through web interface to control connected LEDs[1] using ESP8266 NodeMCU development kit. The client, using a WiFi-compliant device, connects to the subnet of the kit's WiFi module, that runs in Access Point mode and is able to access the web interface using a web client.

This project shows a basic approach, how anyone with a use of an affordable kit can make traditionally non-remotely controlled objects, now remotely controlled. That is to some extent a definition of *Internet of Things* [2] (shortly IoT), the only thing being left is to connect these objects to the Internet.

Firstly the chip itself is introduced, its specifications are listed, then based on that, a design of the resulting embedded system is proposed in section 3. The implementation of both the embedded system and the web interface is briefly described in sections 5 and 6.

# 2 ESP8266 NodeMCU module

The *ESP8266* is a *System on a Chip* (shortly SoC) manufactured by Chinese company *Espressif* [4]. The SoC consists of a 32-bit *Micro Controller Unit* (MCU) and a WiFi transceiver, more about the chip's parameters in subsection 2.1.

ESP8266 is one of the most popular IoT devices available, and therefore there are many different modules based on this SoC, e.g. standalone modules — the *ESP-XX series* by *AI Thinker*, or complete development boards like the module used in this project — *NodeMCU DevKit version 1.0* [6] commonly referred to as version 3 [5].

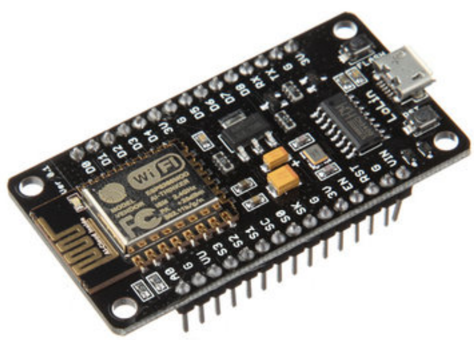


Figure 1: ESP8266 NodeMCU v1.0 (v3)

[1] i. e. LEDs connected to the ESP8266 NodeMCU development kit.

## 2.1 Parameters of ESP8266 SoC

Espressif ESP8266 is a highly integrated Wi-Fi SoC with self-contained Wi-Fi networking capabilities, with low and efficient power usage and compact design, which requires minimal external circuitries.

ESP8266 integrates an enhanced version of Tensilica's L106 32-bit processor with on-chip SRAM, antenna switches, RF balun, power amplifier, low noise receive amplifier, filters and power management modules [7]. Below is listed only a subset of its specifications, only those that are interesting and relevant to this project.

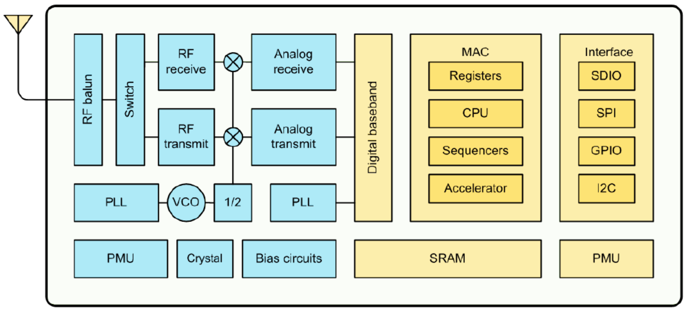



Figure 2: Functional Block Diagram

### 2.1.1 Specifications

| Categories | Items | Parameters |
|---|---|---|
| Wi-Fi | Certification | Wi-Fi Alliance |
| | Protocols | 802.11 b/g/n (HT20) |
| | Frequency Range | $2.4\,G \sim 2.5\,G(2400\,M \sim 2483.5\,M)$ |
| | Output power | +20 dBm in 802.11 b mode |
| | Antenna | PCB Trace, External, IPEX Connector, Ceramic Chip |
| Hardware | CPU | Tensilica L106 32-bit processor, max. clk speed – 160 Mhz |
| | SRAM | SRAM size 64 kB |
| | RAM | RAM size $<$ 50 kB, under normal working conditions |
| | External Flash memory | 512 K normal, up to 16 MB |
| | Clock | internal crystal oscillator, 24 Mhz to 52 Mhz |
| | Peripheral Interface | UART/SDIO/SPI/I2C/I2S/IR Remote Control |
| | | GPIO/ADC/PWM/LED Light and Button |
| | GPIO pins | 17 GPIO pins, multiplexed with various functions |
| | PWM | implement via software, 10-bits @ 1 kHz, up to 14-bit |
| | ADC | 10-bit precision at TOUT (Pin6) |
| | Operating Voltage | $2.5\,V \sim 3.6\,V$ |
| | Operating Current | Average value 80 mA |
| | Standby power consumption | less than 1.0 mW |
| | Operating Temperature Range | $-40°C \sim 125°C$ |
| Software | Wi-Fi Mode | Station/SoftAP/SoftAP+Station |
| | Security | WPA/WPA2 |
| | Network Protocols | IPv4, TCP/UDP/HTTP |
| | User Configuration | AT Instruction Set, Cloud Server, Android/iOS App |

Table 1: Specifications

## 2.2 NodeMCU DevKit V1.0 Development Board

Development boards have all kind of hardware and software features to help developing ESP8266.

As it was stated, one of the development boards used here is the *NodeMCU DevKit v 1.0*. Because of the fact that the hardware is open-source, one can produce its own development boards, therefore there is another naming observation — the producer's version number. The board used here is produced by *LoLin/WeMos* [9] producer and bears a version 3 in their line-up.

### 2.2.1 Parameters of NodeMCU DevKit v1.0 v3

Since second generation, that is *NodeMCU* v 1.0 and *LoLin* version 2, the boards use the newer and enhanced *ESP-12E*. It is claimed that *LoLin* version 3 has more robust USB port, and also 2 reserved pins are now used for `USB power out` and an additional `GND` (see figure 3) [8].

The main features are:

- USB to Serial converter for programming
- 3.3 V regulator for power
- 11 GPIO pins (GPIO 6 – 11 are used to connect the flash memory)
- 4 MB of flash memory
- ADC range from 0 to 3.3 V
- on-board LEDs for debugging
- runs at 3.3 V, and its I/O pins are not 5 V tolerant
- can only source or sink 12 mA per output pin [5]

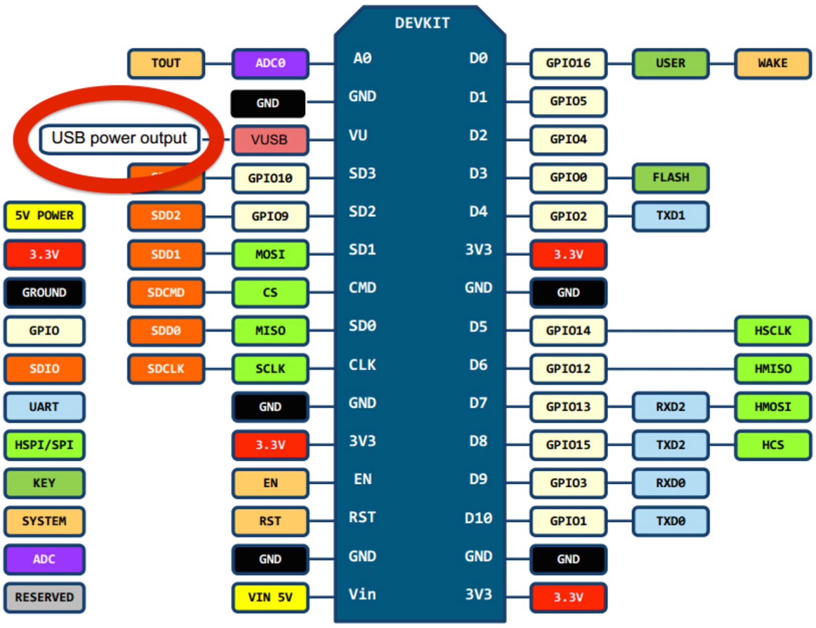



Figure 3: Version 3 differences and Pin mappings

## 3 Design of Embedded System for Wireless LED Control

Firstly we must take into consideration the electrical and material characteristics of the *ESP8266 NodeMCU* module, mainly maximum ratings of input and output supply current, input voltage and recommended operating temperatures. Below are listed all relevant chip data / characteristics, mainly taken from the *Espressif ESP8266EX Datasheet* [7].

a) *ESP8266* is a 3.3 V MCU — its I/O operates at 3.3 V typically.

b) *ESP8266* pins are not 5 V tolerant (more than 3.6 V may kill the chip).

c) *ESP8266* does not support hardware PWM, however software PWM si supported on all digital pins.

d) *ESP8266* has a single analog input with an input range of 0 – 1.0 V[2]. [5]

| Parameters | Min | Typical | Max | Unit |
|---|---|---|---|---|
| Operating Temperature Range | $-40$ | Normal | 125 | °C |
| Working Voltage Value | 2.5 | 3.3 | 3.6 | V |
| $V_{IL}$ | $-0.3$ | - | $0.25\,V_{IO}$[3] | V |
| $V_{IH}$ | $0.75\,V_{IO}$ | | 3.6 | V |
| $V_{OL}$ | - | - | $0.1\,V_{IO}$ | V |
| $V_{OH}$ | $0.8V_{IO}$ | | - | V |
| $I_{MAX}$ | - | - | 12 | mA |

Table 2: ESP8266 Electrical Characteristics

Below is a listed table concerning current consumption of *ESP8266* chip with a few rows listed for an overview of how much the chip itself consumes current during common load. Data in the table below were measured based on 3.3 V supply and 25°C ambient temperature. All the transmitter's measurements are based on 90% duty cycle, continuous transmit mode [10].

| Mode | Typical |
|---|---|
| Transmit 802.11b, CCK[4] 1 Mbps, POUT[5]=+19.5 dBm | 215 mA |
| Transmit 802.11b, CCK 11 Mbps, POUT=+18.5 dBm | 195 mA |
| Receive 802.11b, packet length=1024 byte, -80 dBm | 60 mA |
| Standby | 0.9 mA |
| Deep sleep | 10 $\mu A$ |
| Power save mode DTIM[6] 1 | 1.2 mA |

Table 3: Current Consumption

[2] As it was stated, using NodeMCU DevKit up to 3.3 V.
[3] Input/Output voltage – 3.3 V
[4] Complementary Code Keying
[5] Output Power
[6] Delivery Traffic Indication Message

## 3.1 Design of the Electronic Circuit

Based on the previously listed values, we can determine how much the module itself can sustain, and how many components it can supply.

Strictly for project purposes, the components (here only LEDs) will be driven only by the module itself (no external power supply).

Considering, that in this case the *ESP8266 NodeMCU* is supplied via USB interface, the input current is maximally 500 mA. The chip will run in a Wi-Fi Mode — *SoftAP*, and concurrently *Webserver* will be run. Let us consider that for our purposes, only one or maximally two clients may request the page simultaneously. That should not be a heavy load, therefore we can assume at worst 200 mA for the chip itself. That leaves us with $\sim 300$ mA as an output supply current for other components, but practically maximum drive capacity current of all GPIO pins is $16 \times 12\,mA$ [11].

Let us consider a common LED with following ratings:

| **Parameter** | **Value** |
|---|---|
| Viewing Angle | 120° |
| Lens | 5 mm diameter/frosted/round |
| Emitting Color | Red (620 – 625 nm) |
| Luminous Intensity | 1000 – 2000 mcd |
| Forward Voltage | 2 V – 2.2 V |
| Maximum Current | 20 mA |

Table 4: A “Common” LED ratings

The purpose is to control these LEDs using the module, therefore using only the GPIO pins. Each GPIO pin in output mode can supply maximally 12 mA. Theoretically we can drive up to 25 LEDs, though it should be noted, that this is not the maximum number of LEDs, one can achieve higher number using techniques like *Multiplexing* or *Charlie-plexing*, but again the purpose of this project is to show the ease of individually controlling LEDs.

In reality we can use only 11 GPIO pins. In the end for our purposes here, we will control only 9 LEDs using pins starting from D0 to D8 (skipping the RX and TX pins).

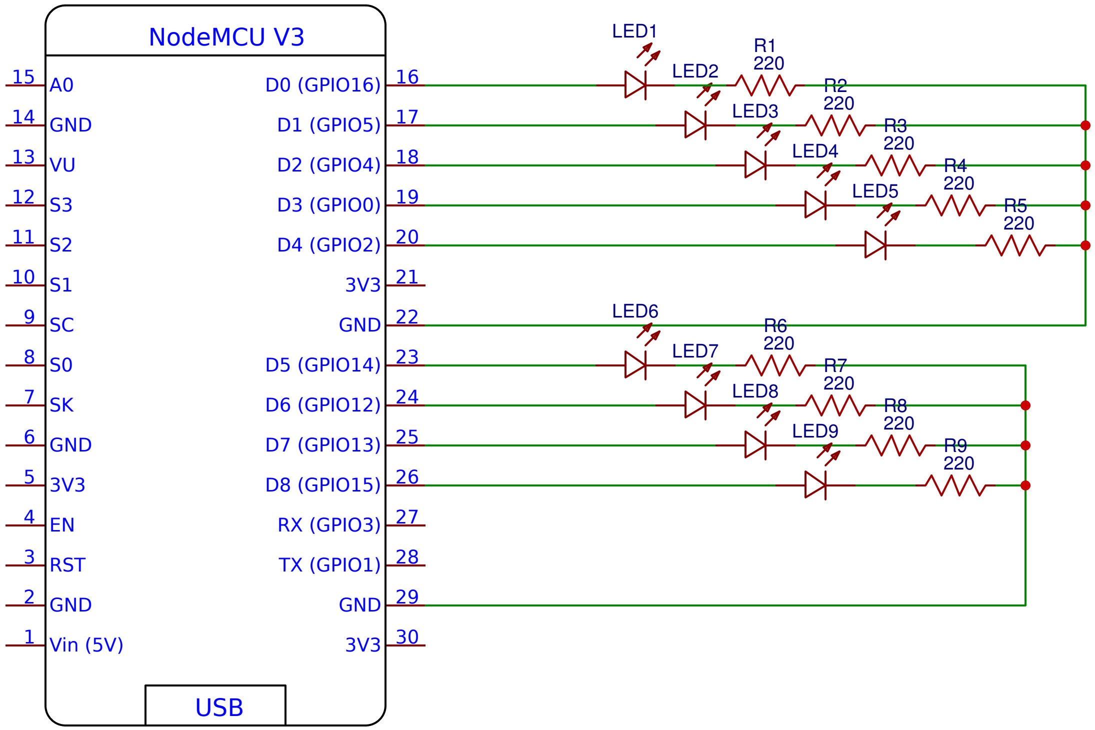
Using Ohm’s law, we determine needed resistance for each LED:

$$R = \frac{V}{I}$$

$$R = \frac{3.3 - 2.0}{12 \times 10^{-3}}$$

$$R = 108\,\Omega$$

Minimum resistance needed for a single LED is $108\,\Omega \pm 5\,\%$. in our case, having only (at this moment) $220\,\Omega$ resistors at hand, then $220\,\Omega$ resistors will be used instead. On the next page, there is the final electrical circuit schematic.




| | | |
|---|---|---|
| TITLE: ESP8266 LED Schematic | | REV: 1.0 |
| Date: 2018-12-21 | | Sheet: 1/1 |
| EasyEDA V5.8.22 | Drawn By: xsmutn13 | |

# 4 Design of the Web Interface

The individual control of LEDs is driven by the on-board processor itself. The client (e.g. an android phone) connects to the *ESP8266*-made wireless network and using a common web client (e.g. *Google Chrome*) sends requests (ideally asynchronous) to the web server, which is also being run on the chip of *ESP8266*.

## 4.1 GUI Requirements

1. The interface should be simple and intuitive, the task is to solely control the LEDs, nothing more.
2. Should be *Android* and *iOS* compliant, as well as on the desktop — therefore using a responsive web design is important.
3. Only a one way of interaction — taps on the screen or clicks, the response should be immediate, both on the web and on the board (here ideally immediate).
4. The overview image should be simple, clear, color themed.
5. User should not be required to write any input, all input should be abstract and predefined with certain constraints and ranges in mind.
6. Should be lightweight both performance speaking and file size speaking (low size implies low bandwidth requirements).
7. On-web requests should be only used for controlling the LEDs, no receiving data and no refreshing of already loaded pages — that is slow, demanding and unnecessary in our case.

## 4.2 Design of LED Controls

Using $3 \times 3$ LED matrix one can come up with a fair number of sequences (rotations) — *Modes* of LEDs and their variations — *Parameters*. All *modes* and *parameters* are summed in table 5.

Switching *ON/OFF* LEDs can be implemented as a matrix of *checkboxes*. Adjusting *parameters* like *speed* can be done using a *slider*. *Direction* can be picked from a list implemented as *radio buttons* (only one *direction* at a time). *Length* as a number of LEDs *ON* simultaneously as a *numeric input*. And finally only one *mode* can be active at a time — as a *toggle* button at each *mode* "section".

| **Mode** | **Modifiable Parameters** |
|---|---|
| Individual | ON / OFF |
| One by one (Sweep) | Speed |
| Row by row | Speed, Direction |
| Column by Column | Speed, Direction |
| Circle (Rotation) | Speed, Length, Direction |
| Swap | Speed |
| Arrow | Speed, Direction |

Table 5: LED Controls

# 5 Implementation of the Embedded System

This section consists of a summary of implementation of the embedded system designed in the previous section 3. The *ESP8266 NodeMCU DevKit* was set up and programmed using the *Arduino IDE* [3] version 1.8.7 and *ESP8266 Arduino Core* library [12] installed via Arduino built-in *Boards Manager* [13].

The project was implemented in *C++* language using mainly *C* language features and *procedural* paradigm, rather then "heavyweight" *C++* features or *object-oriented* paradigm. The code was written with *MISRA-C* [17] guidelines in mind, but is not fully "*MISRA-C*-compliant". For further details about the structure of the project, see section 8.

## 5.1 Access Point Implementation

*ESP8266* SoC runs in a WiFi mode called — *Soft Access Point*. The only difference between classic *Access Point* is that the *ESP8266* cannot connect connected stations to a wired network, because it does not have a wired interface [14]. *SoftAP* mode was implemented using *ESP8266WiFi* library [15], which is part of the *ESP8266 Aruino Core* library. Any requesting station may then connect and is assigned a local IP address using implemented *DHCP* protocol.

## 5.2 Web Server Back-end

Back-end of the web server running on the processor of *ESP8266* was implemented using built-in *ESP8266WebServer* library [16]. It is a simple HTTP server[7] running on a default web server port and on a predefined server's local IP address — `192.168.4.1`.

### 5.2.1 File Management

After a client is connected to the wireless network, it may request the website[8] using *HTTP* protocol. Every requested website file is saved on the module's flash memory and managed using *SPI Flash File System*. *SPIFFS* is specifically designed for chips with limited RAM usage, has a flat structure and is more than sufficient for our web server's file structure [18].

Web server files are kept on the flash in two formats. Whenever a web client accepts encoding in a *Gzip* format, server sends the file using *SPIFFS* in a *Gzip* format, otherwise a "minimal", that means whitespace compressed file in its original format is sent. For uncompressed / compressed sizes comparison of web files see section 7.

## 5.3 LED Controls Implementation

Functionality was divided into functions abstracting each sequence. No sequence runs in a loop, the only loop that is run is the main `loop()` function. The main reason is, to not exceed the *watchdog timers'* limits. Based on the network usage, there might be a need to handle Wi-Fi and TCP requests maximally every 3.2 seconds (for *software watchdog*) [19]. This could be accomplished by using a variant of a *yield* function or *delay* in loops, which take more time to execute. But in our case, there is also a need to handle client requests by web server and the requirement is to do it immediately, so there are no *delay* functions either. The current sequence and all other needed data (*timer (speed)*, *current position*, etc.) is maintained by tiny (8-bit) *state variables*. There is no waiting time before handling the client, but the sequence function execution time, which most of the time sets only values of output pins and returns.

[7] No SSL implemented, for our purposes unnecessary.

[8] See section 6 for the website front-end implementation details.

# 6 Implementation of the Web Interface

Web interface was implemented using common web technologies: *HTML5, CSS3, Javascript*. In order for the interface to be responsive, an open source framework was used — *Framework7*, which is a full featured mobile HTML framework for developing hybrid mobile apps or web apps with iOS and Android native look and feel [20].

The resulting web interface consists of only four files:

- `index.html` — Main and only page of the website in *HTML*. Uses *Framework7' CSS* predefined object classes. Each sequence or *mode* has its own tab with its own *parameters* and an *Active* toggle button, which shows whether a sequence is activated[9] and only one sequence can be active at a time, that is also implied by the fact, that only a one tab can be opened at a time.
- `my_app.js` — Main *Javascript* code, which defines the *Framework7* application object and takes care of sending appropriate requests to the server. Requests are sent asynchronously using *AJAX* with *PUT* method, as paths (arguments) with short names of sequences and parameters, e. g.: `/ind?led=2&state=1`, which means sequence *Individual*, LED number 2 and state 1 as *ON*. Also events are added to the functional objects using both *Javascript* and *Framework7* functions.
- `framework7.min.css` — Consists of *Framework7* predefined styles, and two mine rules — that is the power of this framework.
- `framework7.min.js` — Consists of *Framework7 Javascript* classes defining the behavior of objects, the client interacts with.

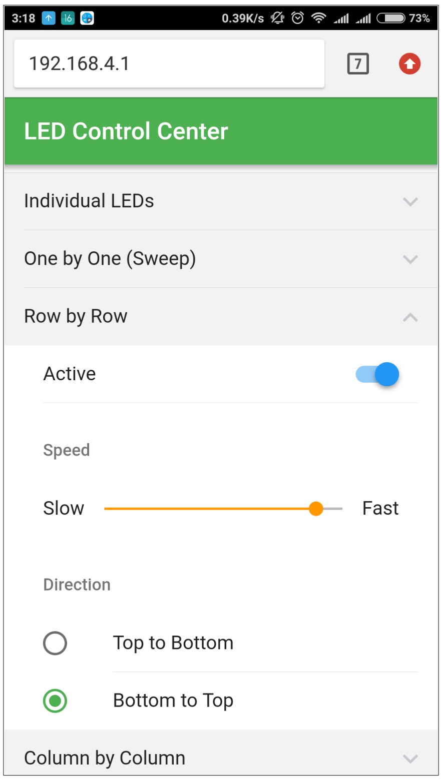



Figure 4: Resulting Web GUI

The resulting video of using the app can be seen at `https://github.com/KentrilDespair/ESP8266-LED-Control-Center`.

[9] If no sequence is activated and no sequence was active previously, then no sequence runs.

# 7 Tests and Results

This sections consists of results of testings, that I made during the implementation and that had an impact one way or another on the performance of the embedded system.

## 7.1 Compression - Impact on Size

During the project, files of the web server were kept in three "formats".

a) Normal — in which it was written.
b) Minimal — without whitespace characters, has `.min` in its name. Used online converter *HTMLCompressor*[10].
c) *Gzip* format — zipped using `$ gzip -k file.min.ext`. *Gzip* parameter `--best` offered only a very slight difference in size (only several bytes), not worth the cost, whereas parameter `--fast` was marginally worse than normal compression.

| **File name** | **Normal size** | **Rel**[11] | **Minimal** | **Rel** | **Compressed** | **Rel** |
|---|---|---|---|---|---|---|
| `index.html` | 19 291 B | 0 % | 9 132 B | 47.34 % | 1 107 B | 5.74 % |
| `my_app.js` | 9 110 B | 0 % | 6 144 B | 67.44 % | 1 205 B | 13.23 % |
| `framework7.min.css` | 937 767 B | 0 % | - | - | 89 829 B | 9.78 % |
| `framework7.min.js` | 547 740 B | 0 % | - | - | 140 334 B | 25.62 % |

Table 6: Compression Comparison

Total size of files in normal format: 1 513 908 B. Total size of compressed files: 232 475 B, which is 6.512 % of the original size.

## 7.2 Compression - Download Time

Comparison of download times using website files in original format, files in minimal "format" and compressed files. Intervals were measure using *Wireshark*, starting from the first request packet, till the last received packet, average value calculated after three tries.

| **Format** | **1 st try** | **2 nd try** | **3 rd try** | **Average Time** |
|---|---|---|---|---|
| Normal | 5.822313 | 5.655180 | 5.716944 | 5.731479 |
| Minimal | 5.542650 | 5.324054 | 5.627228 | ∼5.497977 |
| *Gzip* | 1.313605 | 1.129472 | 1.129969 | ∼1.191015 |

Table 7: Transmission Intervals in seconds

Whenever a client accepts *gzip* encoding, then files are sent in that format, if not then minimal "format" is used instead.

## 7.3 Tweak - Download Unit Size

By default server sends packets with size near to *MTU*, in this case by default — 1460 bytes. Having bigger files, server might send bigger packets, that are going to fragmented and later assembled at the end station, needing in the end more RAM. Increasing a value to 8000 bytes, one can even decrease download times more, which is shown in the table below. [21].

[10] `https://htmlcompressor.com/`
[11] Compression ratio in percentage relative to normal size.

| **Packet Size** | **1 st try** | **2 nd try** | **3 rd try** | **Average Time** |
|---|---|---|---|---|
| 1460 B | 1.313605 | 1.129472 | 1.129969 | ~1.191015 |
| 8000 B | 0.910128 | 0.992827 | 0.955490 | 0.952815 |

Table 8: Transmission Intervals (different sizes) in seconds

### 7.4 Tweak - Wait Time For Connection to Close

Whenever server received many *AJAX GET/PUT* requests, or just sometimes, there was an almost all the time $\sim$ 2 seconds delay before the server's response in a way of switching a LED *ON/OFF*, or there was an undefined behavior in a sequence.

That problem was related to *WiFi* implementation, in which sever waits for the client to close the opened connection for 2000 ms, in most of our cases, this is unnecessarily too long, and with current requests implementation is not practical. Setting a constant `HTTP_MAX_CLOSE_WAIT` defined in `ESP8266WebServer.h` to 10 ms, allowed the server responses to be immediate.

# 8 Enclosed Archive Contents

File structure of the `xsmutn13.zip` archive is described below using a directory tree-like representation.

```
/
├── data
│   ├── framework7.min.css
│   ├── framework7.min.css.gz
│   ├── framework7.min.js
│   ├── framework7.min.js.gz
│   ├── index.min.html
│   ├── index.min.html.gz
│   ├── my_app.min.js
│   └── my_app.min.js.gz
├── definitions.h
├── documentation.pdf
├── leds.h
├── leds.ino
├── main.h
└── main.ino
```

Folder `data` contains files, which are to be uploaded to the *ESP8266* using *Arduino IDE* built-in *SPIFFS* sketch uploader. The files are in a minimal "format", that is without any unnecessary whitespace characters[12], or in *Gzip* compressed formats of these "minimal" representations.

# 9 Conclusion

In conclusion, the project was implemented based on the design defined in section 3. The design requirements reflected the assignment specification and all of the points were successfully implemented. Later on, the system was pushed even further, to be as much user responsive as possible.

[12]It might not be that readable, but because of the maximum upload size of the archive in IS, only "minimal" files could be enclosed. For full versions visit `https://github.com/KentrilDespair/ESP8266-LED-Control-Center`.

# 10 References


[1] *Espressif: ESP8266EX Overview* [online]. Last edited: 2018. [Accessed 22-12-2018]. Available at: https://www.espressif.com/products/hardware/esp8266ex/overview/

[2] *Internet of things* [online]. Last edited: 22-12-2018. [Accessed 22-12-2018]. Wikipedia. Available at: https://en.wikipedia.org/wiki/Internet_of_things

[3] *Arduino Software* [online]. Last edited: 2018. [Accessed 22-12-2018]. Available at: https://www.arduino.cc/en/Main/Software

[4] *Espressif Systems* [online]. Last edited: 2018. [Accessed 22-12-2018]. Available at: https://espressif.com/en/

[5] Pieter P. *A Beginner's Guide to the ESP8266* [online]. Last edited: 08-03-2017. [Accessed 22-12-2018]. Available at: https://tttapa.github.io/ESP8266/Chap01%20-%20ESP8266.html

[6] *NodeMCU DEVKIT V1.0* [online]. Last edited: 10-01-2010. [Accessed 22-12-2018]. Github, Inc. Available at: https://github.com/nodemcu/nodemcu-devkit-v1.0

[7] Espressif Systems. *Espressif ESP8266EX Datasheet* [online]. Version 6.0. Last edited: 11-2018. [Accessed: 22-12-2018]. Available at: https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

[8] Stör M. *Comparison of ESP8266 NodeMCU development boards* [online]. Last edited: 28-09-2015. frightanic.com. [Accessed: 22-12-2018]. Available at: https://frightanic.com/iot/comparison-of-esp8266-nodemcu-development-boards/

[9] *WEMOS* [online]. Last edited: 2018. [Accessed: 22-12-2018]. Available at: https://www.wemos.cc/

[10] *Sleeping the ESP8266* [online]. Last edited: 05-07-2016. ESP8266 Community wiki. [Accessed: 22-12-2018]. Available at: https://www.esp8266.com/wiki/doku.php?id=esp8266_power_usage

[11] *GPIO Maximum current Imax* [online]. Last edited: 17-03-2016. Espressif Discussion Forum. [Accessed: 22-12-2018]. Available at: https://bbs.espressif.com/viewtopic.php?t=139

[12] *Arduino core for ESP8266 WiFi chip* [online]. Last edited: 22-12-2018. Github, Inc. [Accessed: 22-12-2018]. Available at: https://github.com/esp8266/Arduino

[13] Grokhotkov I. *Installing — Boards Manager* [online]. Last edited: 2017. ESP8266 Arduino Core Documentation. [Accessed: 22-12-2018]. Available at: https://arduino-esp8266.readthedocs.io/en/latest/installing.html#boards-manager

[14] Grokhotkov I. *ESP8266WiFi library* [online]. Last edited: 2017. ESP8266 Arduino Core Documentation. [Accessed: 22-12-2018]. Available at: https://arduino-esp8266.readthedocs.io/en/latest/esp8266wifi/readme.html

[15] Arduino. Grokhotkov I. *ESP8266WiFi Source* [online]. Last edited: 18-12-2018. Github, Inc. [Accessed: 22-12-2018]. Available at: https://github.com/esp8266/Arduino/tree/master/libraries/ESP8266WiFi

[16] Grokhotkov I. *ESP8266WebServer Source* [online]. Last edited: 18-12-2018. Github, Inc. [Accessed: 22-12-2018]. Available at: https://github.com/esp8266/Arduino/tree/master/libraries/ESP8266WebServer

[17] *MISRA* [online]. Last edited: 2016. [Accessed: 22-12-2018]. Available at: https://www.misra.org.uk/

[18] Andersson P. *SPIFFS (SPI Flash File System)* [online]. Last edited: 15-10-2017. Github, Inc. [Accessed: 22-12-2018]. Available at: https://github.com/pellepl/spiffs

[19] Grokhotkov I. *FAQ Section Updated about Watchdog resets* [online]. Last edited: 14-11-2018. Github, Inc. [Accessed: 22-12-2018]. Available at: https://github.com/esp8266/Arduino/pull/2533/files

[20] *Framework7* [online]. Last edited: 27-12-2018. [Accessed: 22-12-2018]. Available at: https://framework7.io/

[21] *Esp8266 doesn't send/receive packets with size above 1500 bytes* [online] Last edited: 04-12-2016. Github, Inc. [Accessed: 25-12-2018]. Available at: https://github.com/ARMmbed/esp8266-driver/issues/18